FAI 2024 Final Project Report

EE4, B09602017, Tsung-Min Pai

June 15, 2024

1 Introduction

這大概是我第二次玩德州撲克,基本上是毫無 ruled-based 的想法可以 implement,畢竟我就不是個 expert。但是在理解遊戲玩法後，我認為在不考慮外在因素的情況下，就是一個機率、期望值以及 GTO 的遊戲，當然現實世界中的玩法複雜得多，但我一個什麼都不會的只能先從這個地方著手。

2 Method

除了下面所提到的方法外，因為我們是要比對方還要多籌碼，所以每一個都有用的就是在確定贏夠多的情況下，我就 fold 到底，直接洗盲洗到最後。基本上就是算出現在 fold 到底對方跟自己分別是多少錢就好。

2.1 Winning Rate Evaluator

我分析了一下在兩位玩家的情去下的所有排列組合, 總共有 $C_2^{52}C_2^{50}C_5^{48}$ 種可能, 在 preflop 時會有 $C_2^{50}C_5^{48}$ 種可能，flop 會有 $C_2^{47}C_2^{45}$ 種，都不是單純用 Python (even with multiprocessing, multithreading) 可以在幾秒內窮舉出來的，所以我一開始先嘗試了以下方法:

2.1.1 Monte-carlo

因為我自己認為 python 無法在 5s 內窮舉所有可能性,所以乾脆用一下 monte-carlo,在時間內隨機 sample 最多種可能性，並且在時間快要結束時獲取勝率，可以在一定的 range 內獲得相對準確的勝率估計。

2.1.2 Lookup Table

或是可以透過預先運算建成一個大表，執行時只要查表就可以立刻獲得結果，但在分析此表所需的儲存空間後發現無法在限制內的 500MB 做到。

2.1.3 C++ Multithreading with Subprocess

我使用 C++ 直接重刻一個 Hand Evaluator，再直接窮舉，加上 Multithreading 後執行速度飛快，可以在我本機上花 3 秒左右的時間完成 flop 的運算，甚至不需要 Monte-Carlo，可以不失精確性。綜合以上幾種方法，我決定在 preflop 使用查表的方式，因為考慮同花、不同花的對稱性後可以把整個表縮小到 13x13，但有最大的運算量。我使用 C++ 完成了整個 preflop 的 JSON 表，之後運行就可以直接在 Python O(1) 讀取了。翻牌後我則採用 Real time 的 C++ Subprocess 來做用運算，可以在不用 Monte-Carlo 犧牲準確度的狀況下完成勝率的估計。但這個方法在我寄信問助教發現無法用 C++ Subprocess 之後告吹，小小難過了一下。

2.1.4 Preflop Table

在 2.1.3 中，我建立了 preflop 的 table，而這個並不需要用到 C++ subprocess，因此我在後面的方法都有採用這個 table 做 preflop 的勝率判斷。不管是 rule-based 或是 RL 都有用到。

2.2 Rule-based Probability Decision Agent

我下一步就是基於我建立的勝率表去進行簡單的 decision making，簡單是指邏輯簡單，但實際玩起來我自己的決策都沒辦法贏太多人，因為我根本就不會玩德州撲克，因此我請教了幾個有在打錦標賽的朋友，他們建議我如果要用 if else 去寫機器人，直接玩 AoF 勝率以及可行性還比較大。

2.2.1 Decision Tree

基本上就是依照2.1.3跟2.1.3進行不同 State 用簡單的邏輯做出決策，例如在 Preflop 時勝率 < 50 時就 Fold。並且配合我們的洗盲策略，基本上就可以完勝 Baseline 0-3，由此可知精準的預測其實很重要，但這邊的決策寫法我也試試了很久，偏向工人智慧了。

2.2.2 Raise Agent with Preflop Table

這個跟2.2.2一樣，只是 allin 變成 raise 250 之類的數字，但這個比現並沒有 allin agent 來得好，可能是沒有起到 bluff 的效果，反而弄巧成拙。

2.2.3 AoF Agent with Preflop Table

根據我朋友的建議,我用我建立的 **preflop table 在 preflop 時進行查表的動作,在某個勝率 (threshold) 以上的話我在 preflop 就 allin**, Table 1 顯示了我的 AoF 針對不同 threshold 取得的勝率。可以看到 threshold 45 有最好的平均表現; 但是 42 在某些 baseline 有著 70% 的勝率。並且 Summary 是排除 baseline 0 的總和, 因此最後選擇 **threshold 45**。

Threshold	53	49	45	44	42
Baseline0	0.515	0.570	0.525	0.560	**0.580**
Baseline1	0.475	0.545	0.540	0.625	**0.705**
Baseline2	0.590	0.510	0.585	**0.615**	0.535
Baseline3	0.460	**0.500**	0.410	0.335	0.135
Baseline4	0.475	0.395	0.430	**0.520**	0.440
Baseline5	0.335	0.310	**0.355**	0.295	0.180
Baseline6	0.380	0.395	**0.435**	0.385	0.285
Baseline7	0.420	0.405	**0.440**	0.380	0.225
Summary	3.135	3.060	**3.195**	3.155	2.505

Table 1: **Winning Rate of each threshold of each Baseline.**

2.3 RL-based Agent

由於自己德州撲克的實力不足，其實對於 Baseline5 之後的應對就有點吃力了，因為我純粹寫 State 的判斷的話，遇到詐唬或是其他技巧我基本上就會輸爛，因此我想到我可以用 Baselines 們跟我的 Agent 做互動去 train 個 RL 的模型出來，基本上 ML 不太有可能畢竟我也沒有 Training Data，因此這邊我嘗試了幾個 RL based 的方法。

2.3.1 Offline Q-learning

再來就是第一個想到的 RL 方法，經過查詢後發現 Q-learning 可能蠻適合做這次的任務因此我就試了一下 Q-learning，但效果不是很好，轉而往2.3.2邁進。

2.3.2 Offline DQN

在使用 Q-learning 發現效果不是很好後，我使用了 Deep Q Network 來避免傳統 Q Learning 在面對大型或高維度狀態空間時 Q 值表格過大的問題。但這個我實現起來的效果也沒有很好，因此經過繼續查詢，找到了2.3.3的方法。

2.3.3 MCCRF Agent

再來就是現在主流的解決德州撲克 AI 的方法,Monte Carlo Counterfactual Regret Minimization(MCCFR)，我是看到 DeepStack 和 Libratus 打贏職業牌手的介紹才想說用這個方法試試看，並且在撲克領域，CFR 已被證明是有效的，特別是當使用涉及機會-結果抽樣的特定領域的增強時。這方法確實也是我 **RL-based 裡面的 algorithm 表現最好的**。

- **Counterfactual Regret Minimization(CFR):** CFR 演算法是一種基於**反事實遺憾最小化**的 RL algorithm。在德州撲克中，由於**資訊不完全**，玩家需要根據對手的行為和自己的信念來做決策。CFR 演算法透過計算每一步行動的**遺憾值**（即實際收益與最佳可能收益之差），來引導 agent 進行學習。agent 會根據遺憾值的大小來調整其策略，以最小化整體的遺憾值。

- **Monte Carlo Tree Search:** 蒙特卡羅樹搜尋包含兩層意義，首先它是一種 Tree search 方法，和 BFS, DFS 一樣，它需要對 tree 進行遍歷。其次蒙特卡洛強調了它並不是一種確定性的搜尋演算法，而是透過啟發式的方式，**讓樹朝最優解方向生長，降低搜尋空間，實現了類似 Pruning 的功能**。包括了 selection, expansion, simulation, and back propagation。

- **MCCFR:** CFR 演算法需要遍歷賽局樹，以計算某個動作的遺憾值。然而，當博弈樹的規模非常大時，遍歷整棵樹的成本非常高。一種改進的方法是在單局賽局中對動作進行 Pruning。例如，在某個資訊集上，所有合法動作共有 M 個，演算法可以人為地將搜尋範圍限制為 N 個動作，並且隨著遍歷層級的加深，N 的數值可以逐漸減少。作為補充，**演算法需要增加博弈的次數，以盡可能涵蓋更多的動作**。

3 Comparison

我表現比較好的就是 AoF Agent (2.2.2) 跟 MCCFR model (2.3.3)。對於 AoF 來說，這個動作我自己在玩德撲也會使用，走一個人家不敢跟的路線，然後就賺一輪底池，基本上就是小贏一點，這對於我們這次的作業是非常適合的策略，因為我們不是要賺大錢，只要在最後大於對方的籌碼就算贏。但其實我朋友跟我玩很久之後就知道我的套路不過這個問題在這邊不會發生，因為 baseline 我這樣跑沒有什麼問題，我也不認為其他人會針對這種行為寫應對方法，我覺得很多人可能在 preflop 遇到 allin 就會 fold，除非拿到什麼 AA pair 之類的，但我認為這樣做的勝率其實確實很高。並且如果我在前幾 round allin 賺到夠多錢，後面就洗別人盲洗到他輸，我也跑了一個迴圈去找出有著最好勝率的 threshold

這邊我以 Table 2 來呈現我的兩個主要方法的結果，我是讓他們跟 baseline 打個 1000 場去取得的勝率。

Baseline	1	2	3	4	5	6	7	Summary
AoF	0.540	**0.585**	0.410	**0.430**	0.355	**0.435**	**0.440**	**3.195**
MCCFR-1	**0.580**	0.521	**0.645**	0.148	**0.420**	0.307	0.089	2.710
MCCFR-2	0.305	0.278	0.330	0.090	0.195	0.126	0.068	1.392
Q-learning	0.263	0.326	0.360	0.017	0.134	0.103	0.029	1.232

Table 2: **Winning Rate of AoF and MCCFR of each Baseline.**
(AoF threshold: 45, MCCFR-1 為沒有考慮位置, MCCFR-2 為考慮位置)

可以看到 AoF 表現優於我最好的 RL-based 模型，必且在加入 switch seats 機制後，MCCFR 的表現大幅度下降，我也沒有足夠的時間訓練完整個 agent，因此最後選擇 AoF Agent 作為我比賽的 agent。但其他 agents 也有一併附在 other agent folder。

4 Conclusion

雖然 RL 感覺很適合用在這任務上，因為會希望他學出 Rule-Based 有 Biased 地方，但我看了一下 CMU train 的資源，遠遠不是我一張 4070 train 五天可以比擬的，我這幾天 MCCFR 這樣跑下來大概就模擬了 5000 場左右吧，我認為這對要訓練好一個德州撲克模型是遠遠不夠的，基本上 baseline 3, 7 都跑超久，大幅度地降低訓練的次數。即使使用 offline 的 replay buffer 資源仍然遠遠不夠用。此外，根據特定 baseline 訓練出的模型不一定適合對付其他對手，經常會有 policy 上的漏洞，一旦被發現就會陷入不斷失輸錢狀態。還有一點是 seats 也要考慮進去，我在前三天都沒有把 seats 做交換，結果第四天想到這個問題的時候，performance 直接掉了 $2-30\%$。

綜合以上因素，雖然 RL 模型看起來很強大，但實際效果卻不盡如人意。在最簡單的牌型比較中都存在問題，只能正確判斷高牌和對子的大小，而且容易因為致命弱點被針對而失敗。相比之下，rule-based 的方法每一步都有明確的邏輯，通過多次對戰可以得到更好的 action，且不容易有致命弱點被攻擊。因此，我選擇 rule-based 的 AoF 方法作為我的策略。

其實一開始用 C++ subcessor 做窮舉的效果還不錯，畢竟就是最 rule-based 的 GTO 方法，這其實也是很多線上機器人用的演算法，但可惜後來不能使用，不過可能會繼續研究一下如何讓他變得更好。

References

https://chatgpt.com/c/c3b1be0d-abb3-41b6-8139-2b2bf11303d9

https://icml.cc/media/icml-2019/Slides/4443.pdf

https://app.gtowizard.com

https://blog.csdn.net/qq$_3$6691985/$article/details/116793223$

$http://turingai.ia.ac.cn/share_algorithm/detail/79$

$http://www.aas.net.cn/cn/article/doi/10.16383/j.aas.c210127?viewType=HTML$

$https://www.jiqizhixin.com/articles/2017-03-03$

$https://www.youtube.com/watch?v=DqrcHlZmYDQ$

$https://www.pressplay.cc/project/4D6E2582971ED791FF711B342253E559/about$